cribsheet

April 12, 2020

1 Exam Crib Sheet

1.1 Access modifiers

- `private` → no access from outside this class
- `protected` → access from this class and its subclasses
- `public` → access from anywhere; go nuts

1.2 Collections

1.2.1 `ArrayList`

- length can be changed dynamically
- index starts at zero; goes to length-1

Imports `import Java.util.ArrayList`

Field Declaration `private Arraylist<ElementType>;`

Creation `ArrayList<ElementType> listName = new Arraylist<>();`

Methods

- `ArrayList.clear()` → empty the list
- `ArrayList.add(Element)` → append the `Element` to the list
- `ArrayList.size()` → return the number of elements in the list
- `ArrayList.remove(int index)` → remove the element at index from the list
- `ArrayList.get(int index)` → return the element in the list at index
- `ArrayList.addAll(otherCollection)` → add an entire other collection object to `ArrayList`

1.2.2 `Array`

- fixed-size collection
- can store primitive types and references

No Imports!

- import `Java.util.Arrays` for useful features tho

Field Declaration `String[] shoebox;` → an array of strings

`public shoebox[] = {"words", "words"};` → no length needed; comes from initialized variables

`anArray = int[10]` → holds ten `int`s

`String[][]` → an array of arrays

Access `shoebox[1]` → array index from 0 to n-1

Methods

- Array methods:
  - `Array.length` → *NO PARENTHESES!* returns the length of the array
- Static methods from `Java.util.Arrays`:
  - `Arrays.asList(array);` → a List interface into `array`
  - `Arrays.equals( type array1[], type array2[] );` → returns true if `array1` and `array2` are equal
  - `Arrays.sort(arr);` → sort `arr` into ascending numerical order
  - `Arrays.binarySearch(arr[], key);` → find `key` in `arr[]` by bisection search. `arr[]` must be sorted.
  - `Arrays.fill(arr[], value);` → make every element in `arr` into `value`
- Other:
  - `System.ArrayCopy( source, sourcePos, dest, destPos, length );` → copy `length` elements from `source` to `dest`
  - will go like:
    * `source[sourcePos]`→`dest[destPos]`
    * `source[sourcePos + 1]`→`dest[destPos + 1]`
    * ...
    * `source[sourcePos + length - 1]` → `dest[destPos + length - 1]`
    * elements in `dest` before `destPos` are not affected

1.2.3 `HashMap`

- a simple database based on key/value mappings
- need to declare a key type and a value type
- unidirectional: you can look up a value with a key but not a key from a value

Imports

- `import java.util.HashMap;`

Field Declaration

Creation `HashMap<keyType, valueType> hm = new HashMap<>();`

Methods

- `hm.put(Key, Value)` → add a new key/value pair to the map
- `hm.get(Key)` → return the value associated to `Key` in the map

1.2.4 `HashSet`

- a list with no duplicates
- not necessarily in order

Imports

- `import java.util.HashSet;`

Field Declaration

- `HashSet<ElementType> hs = new HashSet<>();`

Methods

- `hs.add(Object)` → add a new object to the set

1.3 Iteration

1.3.1 `while`

Syntax

[ ]:
```
while (some_condition) // condition must evaluate to boolean true
{
    do_something;
    change_condition_to_avoid_an_infinite_loop;
}
```

Use when:

- the index in a collection is important
- you don't have a collection at all
- you need to process part of a collection

1.3.2 `for`

Syntax

[ ]:
```
for ( initialization; loop_if_true; post_body_action )
{
    do_the_thing;
    possible_do_the_other_thing;
}
```

1.3.3 For:each Loop

Syntax

```
[ ]: for (ElementType elementName : collection)
     {
         do_this_on_each_collection_element;

     }
```

1.3.4 Iterator

Syntax

```
[ ]: for (Iterator<ElementType> it = someCollection.iterator(); it.hasNext(); ) //␣
      ↪return iterator object
     {
         do_something( it.next() ); // .next will return the next and advance
     }
```

Iterator methods

- `iterator.hasNext()` → true if we can keep iterating
- `iterator.next()` → returns the next object in the collection
- `iterator.remove()` → removes the last object from the collection

Use when:

- you need to remove items from a collection object

1.4 jUnit Basics

Test Annotations

- `@Before` - run before each `@Test` method
- `@After` - run after each `@Test` method
- `@Test` - a test case

Asserts

- `import org.junit.Assert.*`
- test that your code did the right thing
  - `assertEquals(String message, obj1, obj2)` → test passes if `obj1` and `obj2` are equal; give `message` if not
  - `assertTrue(String message, obj)` → test passes if `obj` booleans to True
  - `assertFalse(String message, obj)` → test passes is `obj` booleans to False

1.5 Inheritance

1.5.1 Basic Syntax

To inherit from another class:

[ ]:
```
public class Animal // superclass
{
    public Animal()
    {
        do_animal_stuff();
    }
}

public class Bird extends Animal // a subclass
{
    public Bird()
    {
        super();
        do_bird_stuff();
    }
}
```

1.5.2 Polymorphism

- an object of a superclass can be used directly as an object of:
  - the superclass
  - any subclass of the superclass
    - a method needing `Animal` may take `Bird`
    - a method needing `Bird` cannot take `Animal`

`instanceof`

- identify whether an object of of a given type or a subtype thereof

1.5.3 Overriding

- Java looks for methods from subclass to superclass; ends up at `Object`
- to override a method, give a subclass a method with the same name (this is especially useful in `.equals()`)

[22]:
```
public class Animal
{
    private String name;

    public Animal( String name )
    {
        this.name = name; // all animals have names
    }

    public String getName()
    {
        return this.name;
    }
    public void speak()
    {
        System.out.println("<silence>");
    }
}

public class Cat extends Animal
{
    public Cat( String name )
    {
        super(name);
    }

    public void speak()
    {
        System.out.println("Meow");
    }
}
```

[23]:
```
Animal a = new Animal("a");
```

[24]:
```
a.speak();
```

```
<soft rustling>
```

[25]:
```
Cat sylvester = new Cat("Sylvester");
```

[26]:
```
sylvester.speak();
```

```
Meow
```

1.5.4 Abstract classes

- cannot create class instances
- can be inherited from
- have method signatures but not bodies

1.5.5 Interfaces

- a sort of multiple inheritance
- a set of method prototypes which will let you interact with an object
  - these prototypes do not have method bodies
- inherited by `implements` keyword

`default` methods

- these do have method bodies
- methods are inherited by all `implements`ing classes

[ ]:
```
public class Vehicle
{}

public interface License
{}

public class Bicycle extends Vehicle
{}

public class Car extends Vehicle implements License //like a vehicle but also␣
 ↪with a license
{}
```

1.6 Error Handling

1.6.1 Exceptions

Creation

- just a class that inherits from `Exception`

[ ]:
```
public class myException extends Exception
{
    public myException() //constructor
    {}
}
```

Throwing

- javadoc: `@throws`
- function signature for checked exception in methods:

[ ]:
```
public void stage() throws myException
{
    if (numStages = 0)
    {
        throw new myException("narf");
    }
}
```

Exception checking

- check if some function that should nominally work might have a problem
- catch particular exception types

[ ]:
```
try
{
    do_something();
}
```

```
catch ( myException ex )
{
    fix_exception(e); // e is an exception object
}
finally
{
    something_that_happens_last_every_time();
}
```